Hyper-Threaded Multiplier for HECC

Gabriel GALLIN[1,2] and Arnaud TISSERAND[2]
[1]CNRS, IRISA UMR 6074, INRIA Centre Rennes - Bretagne Atlantique, University Rennes 1, Lannion, France
[2]CNRS, Lab-STICC UMR 6285, University South Brittany, Lorient, France.
gabriel.gallin@irisa.fr , arnaud.tisserand@univ-ubs.fr

***Abstract*—Modular multiplication is the most costly and common operation in hyper-elliptic curve cryptography. Over prime fields, it uses dependent partial products and reduction steps. These dependencies make FPGA implementations with fully pipelined DSP blocks difficult to optimize. We propose a new multiplier architecture with hyper-threaded capabilities. Several independent multiplications are handled in parallel for efficiently filling the pipeline and overlapping internal latencies by independent computations. It increases the silicon efficiency and leads to a better area / computation time trade-off than current state of the art. We use this hyper-threaded multiplier into small accelerators for hyper-elliptic curve cryptography in embedded systems.**

I. Introduction

Nowadays, numerous applications require strong security levels on small hardware devices. Public-key cryptography (PKC) is mandatory for providing key exchange and digital signature. RSA, the first PKC standard, is too costly for embedded applications, more than 2000-bit keys are currently recommended for an average security level. *Elliptic Curve Cryptography* (ECC [1]) and *Hyper-Elliptic Curve Cryptography* (HECC [2]) are known to provide a given security level at a lower cost than RSA. For instance, 226-bit ECC offers similar security than 2048-bit RSA. Then ECC and HECC are promoted for implementations in embedded systems.

Efficient (H)ECC hardware accelerators require efficient arithmetic units over finite fields. The most common and costly finite field operation in (H)ECC is the *modular multiplication* (MM) (see Sec. II). Various MM algorithms have been proposed for hardware implementations (see Sec. III). For instance, depending on the width of field elements and type of algorithm and architecture, one $\mathbb{F}_{\mathcal{P}}$ multiplication requires from 30 to 100 clock cycles for (H)ECC typical field sizes. In Sec. IV, we propose a new MM architecture over $\mathbb{F}_{\mathcal{P}}$ with generic prime, leading to efficient pipelined implementations in FPGAs. Sec. V provides implementation details and comparisons to similar MM from state of the art. Sec. VI reports comparisons for some HECC application. Sec. VII concludes the paper.

II. HECC Context

Recent research works have pointed out HECC as an attractive alternative to ECC. HECC is based on a different kind of curves, which allows the size of field elements to be halved, but at the expense of an increased number of finite field operations. In CHES 2016 paper [3], Renes *et al.* presented software implementations of key exchange and signature schemes based on HECC and Kummer surfaces over $\mathbb{F}_{\mathcal{P}}$. They targeted embedded processors ARM Cortex M0 and AVR ATmega. Their results show interesting speedups compared to state of the art for ECC implementations with similar security: 30 % for Diffie-Hellman key exchange and up to 70 % for signature.

HECC involves more field level operations than ECC for each key bit during scalar multiplications. However, one can observe that in ECC most of $\mathbb{F}_{\mathcal{P}}$ operations in point addition/doubling are dependent and must be mostly executed in a sequential way. Then, internal parallelism in ECC is quite limited compared to HECC. For instance, the formulas presented in [3] for HECC show regular patterns of 4 up to 8 independent MM feasible in parallel during scalar multiplication iterations (see Fig. 4). As a comparison, ECC allows 2 MMs to be performed in parallel on average.

The software solution in [3] was optimized for the Mersenne prime $2^{127}-1$. In hardware, dealing with such a specific prime value may lead to limited circuit lifetime and applications. Then, we chose to design units for generic primes $\mathcal{P}$.

III. Background on $\mathbb{F}_{\mathcal{P}}$ Finite Field Multipliers

In [4], Montgomery presented an efficient algorithm for MM over $\mathbb{F}_{\mathcal{P}}$ now called *Montgomery modular multiplication* (MMM). Many algorithms have been derived from this paper. They all improve efficiency by interleaving partial products generation and modular reduction steps to reduce the width of intermediate data and to gain some speedup. One famous variant is the *Coarsely Integrated Operand Scanning* (CIOS) method presented by Koc *et al.* in [5]. However, besides these improvements, Montgomery multiplication still suffers from strong dependencies inside the main loop of partial products accumulation and modular reduction.

Most of modern FPGAs embed many dedicated hardware resources for performing small integer multiplications and accumulations (e.g. $18\times18\pm48$ bits). They are too small for cryptographic operands but they act as efficient basic blocks for building $\mathbb{F}_{\mathcal{P}}$ operators. In order to reach high frequencies, DSP blocks must use several internal pipeline stages. For instance, a typical 3-stage pipelined DSP block forces to wait 3 cycles before reusing any result from the DSP block. Due to data dependencies, one cannot feed efficiently this pipeline during MMM iterations with strong dependencies between iterations. This reduces circuit efficiency with lower utilization of DSP blocks.

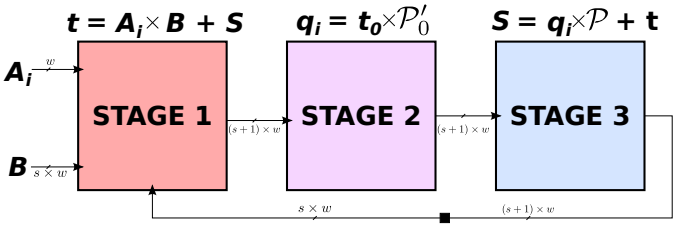

Fig. 1. HTMM decomposition into stages.

In [6], Ma *et al.* proposed a MMM implementation based on Orup algorithm [7]. This implementation is known to be one of the fastest FPGA implementations. However, to get rid of some of internal dependencies, their method implies overheads in terms of size of intermediate data, increasing circuit area (see Sec. V). Many previous works reduced data dependencies by unrolling internal loops. These works require dozens up to hundreds of DSP blocks which is probably too costly for most of embedded solutions.

IV. Hyper-Threaded Multiplier

We chose a different approach to improve efficiency in small hardware MM units. We decided to use the classical CIOS method as a basis to design an *hyper-threaded modular multiplier* (HTMM). Hyper-threading consists in *interleaving* independent operations in the same hardware resource to fill the pipeline as much as possible. HTMM can be seen as multiple logical multipliers sharing the same physical resource. The CIOS method is described in Algo. 1 where $\mathcal{P}$ is a prime modulus and n is the operand width. Each operand is split into s words of w bits such that $n = s \times w$.

In our target accelerators, we do not use internal communications of n-bit words (for $\mathbb{F}_{\mathcal{P}}$ elements) at each clock cycle. Both operands and results of arithmetic units (including HTMM) are transmitted using s words of w bits. Targeted DSP blocks are at most 17-bit wide for one unsigned operand. Using multiples of 17 bits for w saves area and power since we can avoid internal interfaces to accommodate n-bit field elements computed on 17-bit DSP blocks (here $w = 2\times17 \ll n$). Using more than 2×17 bits for w is possible but leads to much larger architectures (this is not interesting for our applications in embedded systems).

CIOS, cf. Algo. 1, performs iterations over words of multiplier operand A. Each iteration corresponds to: partial product of multiplicand B by word A_i (Algo.1:lines 4–10); "quotient" q_i determination from partial product $A_i \times B$ (Algo.1:line 11); and finally addition of $A_i \times B$ with product $q_i \times \mathcal{P}$ (Algo.1:lines 12–19) for reduction step. Reduction of partial product is done by discarding least significant word of t. This partial reduced result is added to the next partial product $A_{i+1} \times B$ during next iteration. Each iteration is thus based on 3 strongly dependent products, and iterations themselves are sequential.

HTMM is based on 3 hardware *stages* as depicted in Fig. 1. Each one corresponds to one step of the iteration described above. *STAGE 1* performs the partial product of one w-bit word of A and s words of w bits from B and the addition with the previous reduced partial product from STAGE 3. "Quotient" q_i is computed in *STAGE 2*. Finally in *STAGE 3*, the product $q_i \times \mathcal{P}$ is performed and added to the product $A_i \times B$ computed in STAGE 1, where the least significant w-bit word of the result is discarded.

Each stage uses fully pipelined DSP blocks to compute partial products at a high frequency. On our target FPGAs (see Sec. V), using the 3 internal pipeline registers of each DSP block leads to 300 to 400 MHz frequencies instead of about 100 to 200 MHz without full pipeline.

When computing a single MM on a pair of operands $(A^{(0)}, B^{(0)})$, the most significant word at iteration i will be available after some delay due to internal pipeline registers. Then it cannot be used directly into the next iteration $(i + 1)$. Hyper-threading use the idle stages to compute other independent MMs in parallel. In our

Algorithm 1 CIOS algorithm (from [5]).

Require: $\mathcal{R} = 2^n$ such as $4\times\mathcal{P} < \mathcal{R}$, $\mathcal{P}' = -\mathcal{P}^{-1} \pmod w$ and A, B two integers such that $0 \le A, B < 2 \times \mathcal{P}$
Ensure: $0 \le S < 2^w$, $0 \le C < 2^w$, $t \equiv (A \times B \times \mathcal{R}^{-1}) \pmod{\mathcal{P}}$, $0 \le t < 2 \times \mathcal{P}$

1: **for** $i = 0$ **to** $s - 1$ **do**
2: $t \leftarrow 0$
3: $C \leftarrow 0$
4: **for** $j = 0$ **to** $s - 1$ **do**
5: $(C, S) \leftarrow t_j + A_i \times B_j + C$
6: $t_j \leftarrow S$
7: **end for**
8: $(C, S) \leftarrow t_s + C$
9: $t_s \leftarrow S$
10: $t_{s+1} \leftarrow C$
11: $q_i \leftarrow t_0 + m \times \mathcal{P}' \pmod w$
12: $(C, S) \leftarrow t_0 + q_i \times \mathcal{P}_0$
13: **for** $j = 1$ **to** $s - 1$ **do**
14: $(C, S) \leftarrow t_j + q_i \times \mathcal{P}_j + C$
15: $t_{j-1} \leftarrow S$
16: **end for**
17: $(C, S) \leftarrow t_s + C$
18: $t_{s-1} \leftarrow S$
19: $t_s \leftarrow t_{s+1} + C$
20: **end for**
21: **return** t

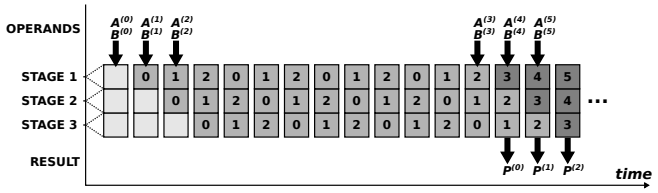

Fig. 2. HTMM usage to compute 3 independent MMs.

HTMM, one can enter 3 independent pairs of operands $\{(A^{(0)}, B^{(0)}), (A^{(1)}, B^{(1)}), (A^{(2)}, B^{(2)})\}$ before the first product $P^{(0)} = A^{(0)} \times B^{(0)}$ is computed. This way, all the stages are full after the very first latency, as illustrated by Fig. 2. Our HTMM can also receive a new pair of operand (e.g. $(A^{(3)}, B^{(3)})$) slightly before the oldest product (e.g. $P^{(0)}$) is transmitted on the unit output (see right part of Fig. 2).

In the future, we will investigate the use of other numbers of pairs of operands (it seems that 3 is probably the most efficient one for this size of field).

V. Implementation and Comparisons

We implemented 2 HTMM versions on 3 different Xilinx FPGAs: Virtex 5 XC5LX110T (V5), Virtex 4 XC4VLX100 (V4) and Spartan 6 XC6SLX75 (S6). Version HTMM_BRAM uses dedicated hardware RAM blocks (BRAMs) to store the 3 pairs of operands. Version HTMM_DRAM uses distributed memory in programmable LUTs (lookup tables) of logic slices to store the operands and all intermediate values (no block RAM).

We implemented MM from [6] to help comparisons on the target FPGAs. We obtained very close results for both area and timing: e.g. 37 DSP blocks for 256-bit $\mathbb{F}_{\mathcal{P}}$ for us and [6].

In order to reduce the iterations number without increasing too much the area, we chose $w = 34$ bits. In HECC, the currently recommended width for $\mathbb{F}_{\mathcal{P}}$ elements is at least 128 bits. Then, we set $s = 4$ words and $n = 4 \times 34 = 136$ bits. In the future, we will study how to use "rectangular" DSP blocks (e.g. 17×24 bits).

Fig. 3 details the architecture of a 128-bit HTMM with configurations of DSP blocks and pipeline registers. Boxes colors correspond to stages of Fig. 1 and Algo. 1. Stages 1/2/3 respectively require 4/3/4 DSP blocks.

HTMM_BRAM, HTMM_DRAM and MM from [6] have been implemented in VHDL using Xilinx CAD tools ISE and iSim 14.7. Tab. I reports implementation results in terms of FPGA resources – number of DSP blocks, BRAMs, FFs (flip-flops), LUTs, slices –, and performances – clock cycles, computation time – for 3 independent MMs. These results correspond to the best place and route (PAR) strategies found after 100 runs of SmartXplorer.

The ECC architecture presented in [6] uses a single MM operator without any BRAM in the unit. But it uses more BRAMs to store large field element in Montgomery domain for Orup optimization (with full width communications in the architecture). In parallel accelerators for HECC, we use small internal communications (w-bit words) and BRAMs in arithmetic units to store various operands and reduce the bandwidth to the central memory. We then consider them as part of the hardware cost (corresponding values are indicated by a * in Tab. I).

For one single MM, both HTMM versions are less efficient than [6]: 69 clock cycles for HTMM versus 25 for [6]. For 3 independent MMs, HTMM requires 79 cycles against 65 cycles for [6] but with almost half the number of DSP blocks and much higher frequency (due to smaller internal values).

Finally, HTMM leads to faster products for about half the cost of the best state of the art solution (Tab. I). On S6 FPGA, HTMM_BRAM leads to 15 % speedup, 48 % DSP blocks reduction, 66 % BRAMs reduction and 33 % slices reduction for 3 independent MMs. HTMM_DRAM provides the same speedup and DSP blocks reduction with only 10 % slices reduction but without any BRAM.

VI. HTMM Usage in HECC based on μKummer

HECC is considered as an interesting solution for PKC on low-cost processors using solutions based on Kummer surfaces [8]. The recent paper [3] presented μKummer, a very efficient software implementation on low-cost devices (8-bit AVR and 32-bit ARM microcontrollers), with 30 to 70 % speedup for key exchange and digital signature.

We work on the design and prototyping of hardware accelerators for PKC (with several curve based solutions) in FPGAs and ASICs. We implemented the μKummer HECC solution proposed in [3] on several FPGAs to evaluate how it performs in hardware compared to efficient ECC solutions from the state of the art.

Fig. 4 shows the amount of parallelism in the combined point doubling-addition operation proposed in [3] (a/s are additions/subtractions and M/S are multiplications/squares in field $\mathbb{F}_{\mathcal{P}}$).

We designed several HECC hardware accelerators using our HTMM unit and various configurations of the architecture (see [9] for more details). We compared the 256-bit ECC solution from [6] and 2 of our accelerators for μKummer HECC (named H1 and H2). H1/H2 accelerators have the following configurations:

- H1: 1 HTMM, 1 $\mathbb{F}_{\mathcal{P}}$ adder-subtractor, 1 CSWAP unit (in charge of key management), 1 output register (for results of scalar multiplications), 1 global memory, 1 global control and 1 program memory;
- H2: H1 configuration with 2 HTMMs and 2 $\mathbb{F}_{\mathcal{P}}$ adders-subtractors (instead of 1 unit arithmetic of each type).

One can notice that a HTMM unit handles 3 MMs in parallel (6 with 2 HTMMs). Architecture of accelerator H1 is depicted in Fig. 5. Internal communications are handled by (de)-multiplexors driven by the control program. Accelerator H2 is similar with more arithmetic units.

Both H1 and H2 accelerators for HECC have been fully implemented on V4 and V5 FPGAs for comparison to the ECC solution proposed in [6] (known to be the fastest ECC solution on FPGA without using hundreds of DSP blocks). The corresponding results are reported in Tab. II. On V5 FPGA, accelerator H1 achieves area reduction by 70 % DSP blocks, 30 % BRAMs and 49 % slices but with a 30 % duration increase compared to ECC. Accelerator H2 (with 2 arithmetic units of each type) reduces both duration by 15 % and area by 40 % DSP blocks, 10 % BRAMs and 10 % slices.

Our results confirm the interest for HECC compare to ECC: either faster primitives for the same area or halve of the area for a similar computation time. HTMM helps to design parallel accelerators without increasing too much the internal

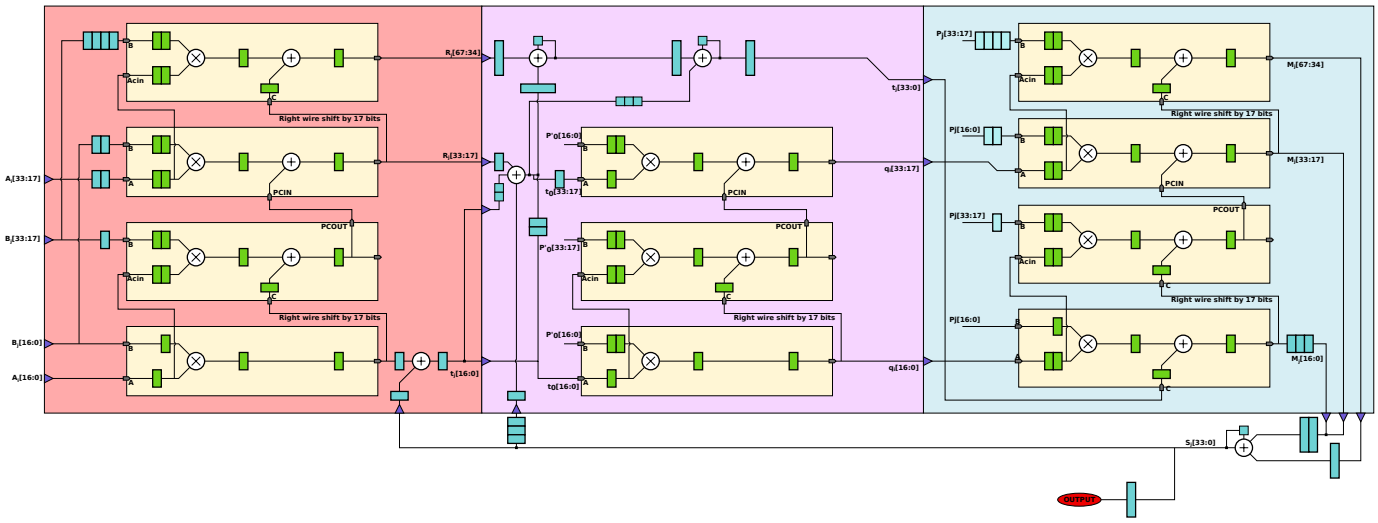

Fig. 3. HTMM architecture details for 128-bit $\mathbb{F}_{\mathcal{P}}$ operands

TABLE I
FPGAs IMPLEMENTATION RESULTS: HARDWARE COST AND PERFORMANCES FOR 3 INDEPENDENT MMs.

Version	FPGA	DSP	BRAM 18K/9K	FF	LUT	Slices	Frequency [MHz]	Clock cycles	Time [ns]
MM from [6]	V4	21	6*/0	1311	1201	879	252	65	258
	V5	21	6*/0	1310	1027	406	296		220
	S6	21	0/6*	1280	1600	540	210		309
HTMM_DRAM	V4	11	0/0	1638	1128	1346	330	79	239
	V5	11	0/0	1616	652	517	400		198
	S6	11	0/0	1631	1344	483	302		261
HTMM_BRAM	V4	11	2/0	615	364	449	328	79	241
	V5	11	2/0	593	371	249	357		221
	S6	11	0/2	587	359	180	304		260

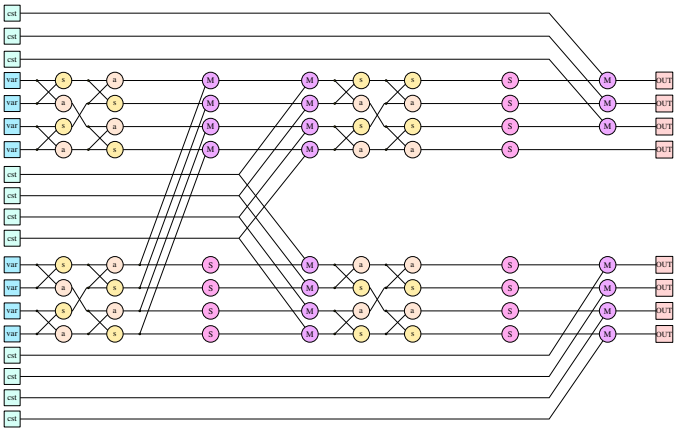

Fig. 4. Computation in combined doubling-addition in μKummer from [3].

communications inside the architecture. Using more than 2 HTMM units for μKummer is currently difficult due to the central data memory.

VII. CONCLUSION

We proposed an hyper-threaded modular multiplier for HECC over 128-bit generic prime fields. It leads to better area / computation time trade-offs than the best state of the art solutions. By improving the hardware efficiency (more hardwired resources are active at each clock cycle), we are able to be slightly faster but a lot smaller than [6] when at 3 independent modular multiplications are computed in parallel.

In the future, we plan to study other HTMM versions. We will also study hyper-threaded schemes impact on energy consumption and on side-channel leakage.

TABLE II
IMPLEMENTATION RESULTS FOR H1/H2 HECC, AND ECC (FROM [6]) ACCELERATORS ON V4 AND V5 FPGAs.

FPGA	Accelerator	DSP Blocks	BRAM 18K	Slices	Freq. MHz	Time ms
V4	ECC	37	11	4655	250	0.44
	H1	11	7	1413	330	0.55
	H2	22	9	2356	330	0.35
V5	ECC	37	10	1725	291	0.38
	H1	11	7	873	360	0.51
	H2	22	9	1542	360	0.32

ACKNOWLEDGMENT

This work was partly funded by the HAH project (Labex CominLab and Lebesgue, Brittany Region, http://www.h-a-h.cominlabs.ueb.eu/).

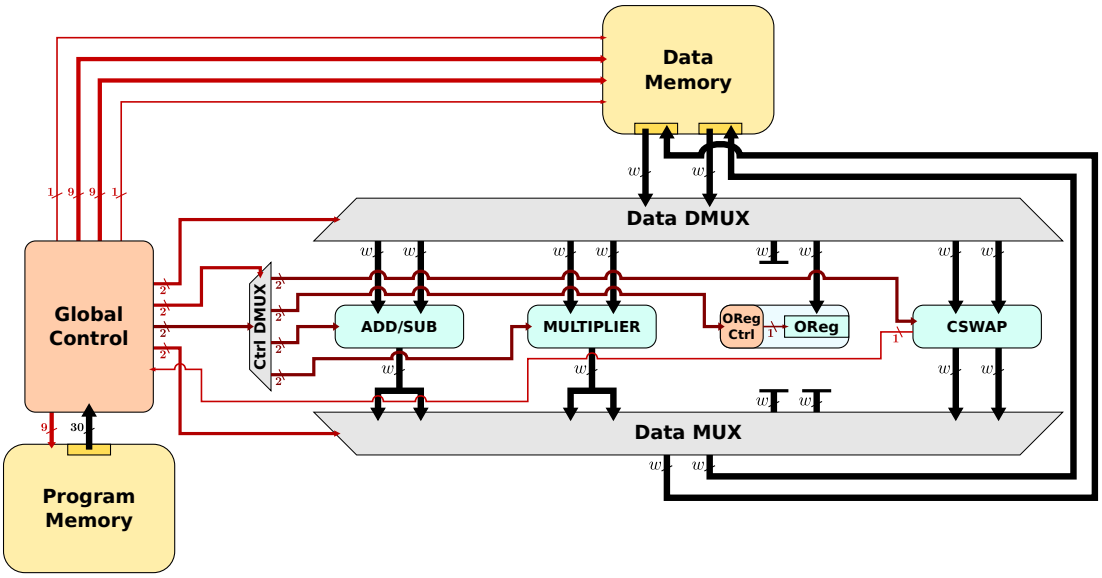

Fig. 5. Hardware accelerator for HECC adapted from the μKummer solution proposed in [3] for software implementation.

REFERENCES

[1] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[2] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, ser. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, Jul. 2005.

[3] J. Renes, P. Schwabe, B. Smith, and L. Batina, "μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers," in *Proc. Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 9813. Springer, Aug. 2016, pp. 301–320.

[4] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-1985-0777282-X

[5] Ç. K. Koç, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing Montgomery multiplication algorithms," *Micro, IEEE*, vol. 16, no. 3, pp. 26–33, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1109/40.502403

[6] Y. Ma, Z. Liu, W. Pan, and J. Jing, "A high-speed elliptic curve cryptographic processor for generic curves over GF(p)," in *Proc. 20th International Workshop on Selected Areas in Cryptography (SAC)*, ser. LNCS, vol. 8282. Springer, Aug. 2013, pp. 421–437. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43414-7_21

[7] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proc. 12th Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, Jul. 1995, pp. 193–199. [Online]. Available: http://dx.doi.org/10.1109/ARITH.1995.465359

[8] P. Gaudry, "Fast genus 2 arithmetic based on theta functions," *Journal of Mathematical Cryptology*, vol. 1, no. 3, pp. 243–265, 2007.

[9] G. Gallin, T. U. Celik, and A. Tisserand, "Architecture level optimizations for Kummer based HECC on FPGAs," in *Proc. 18th International Conference on Cryptology in India (IndoCrypt)*, Chennai, India, Dec. 2017.